CMPT 777 Programming Assignment 2

Sihui Wang 301474102

1. Problem Formulation:

1.1 Notations:

Sudoku Matrix Suppose that the Sudoku solution is represented by a 9×9 matrix, $M \in \mathbb{Z}^{9\times9}$, where m_{ij} denotes the element in the i-th row and the j-th column in M.

Block Representation of Sudoku Matrix M has a block matrix representation:

$$M = \begin{pmatrix} M_{11} & M_{12} & M_{13} \\ M_{21} & M_{22} & M_{23} \\ M_{31} & M_{32} & M_{33} \end{pmatrix}$$

where $M_{ij} \in \mathbb{Z}^{3\times3}(1 \leq i, j \leq 3)$ are 3×3 submatrices.

Matrix of Known Elements We also have a matrix $D \in \mathbb{Z}^{9\times9}$ representing the input known elements. d_{ij} denotes the elements in the i-th row and the j-th column in D. $d_{ij} = 0$ means that the element in the i-th row and the j-th column in M is unknown; while $d_{ij} = k(1 \leq k \leq 9)$ means that the element in the i-th row and the j-th column in M is known, that is, M has the constraint that $m_{ij} = d_{ij}$ if $d_{ij} \neq 0$.

1.2 Constraints and Problem Formulations:

Then, based on Sudoku`s rules, we have the following constraints for $m_{ij}(1 \leq i, j \leq 9)$:

Range Contraints. Each element should fall in the range from 1 to 9:

$$\forall 1 \leq i, j \leq 9, 1 \leq m_{ij} \leq 9$$

Distinction Constraints. Every element in the same row should be distinctive from each other.

For each row i, we have:

$$\forall m_{ij}, \forall m_{ij'}, j \neq j' \rightarrow m_{ij} \neq m_{ij'}$$

Every element in the same column should be distinctive from each other:

For each column j, we have:

$$\forall m_{ij}, \forall m_{i'j}, i \neq i' \rightarrow m_{ij} \neq m_{i'j}$$

Every element in the same block should be distinctive from each other:

For each block $M_{ij}(1 \leq i, j \leq 3)$, we have:

$$\forall m_{pq} \in M_{ij}, \forall m_{rs} \in M_{ij}, p \neq r \vee q \neq s \rightarrow m_{pq} \neq m_{rs}$$

Constraints of Known Elements. If the element in the i-th row and the j-th column is given in the input, then m_{ij} should follow the constraint given by d_{ij}:

$$\forall d_{ij}, d_{ij} \neq 0 \rightarrow m_{ij} = d_{ij}$$

$m_{ij}(1 \leq i,j \leq 9)$ are solutions to the Sudoku problem, if and only if they satisfy all the above constraints. So, we can formulate the Sudoku problem as the satisfiability problem and use Z3 solver to find the solutions.

2. Program Design:

2.1 Initialization:

First, we create the context and the solver:

```
Context ctx=new Context();
Solver solver=ctx.mkSolver();
```

Then, we create a 2D 9×9 array of integer variables, `p`. Here, `p` corresponds to matrix M in the above formulation.

```
IntExpr[][] p=new IntExpr[9][9];
```

Since in python, arrays are indexed from 0, here `p[i][j]`$(0 \leq i \leq 8, 0 \leq j \leq 8)$ corresponds to $m_{i+1,j+1}$ in M.

Next, we write loops to initialize all `p[i][j]`:

```
p[i][j]=ctx.mkIntConst("p_"+i+"_"+j);
```

2.2 Adding Range Constraints:

We write loops to impose range constraints for each `p[i][j]`:

```
solver.add(ctx.mkAnd(ctx.mkLe(ctx.mkInt(1),p[i][j]),ctx.mkLe(p[i][j],ctx.mkInt(9))));
```

Since step 2.1 and step 2.2 both require us to loop over variables i and j, in implementation we can put the code for variable initialization and range constraints in the same loop block.

```
for(int i=0;i<9;i++){
      for(int j=0;j<9;j++){
          p[i][j]=ctx.mkIntConst("p_"+i+"_"+j);
          solver.add(ctx.mkAnd(ctx.mkLe(ctx.mkInt(1),p[i][j]),ctx.mkLe(p[i][j],ctx.mkInt(9))));
      }
}
```

2.3 Adding Distinction Constraints:

For distinction constraints, in implementation we can write a loop over i and impose the distinction constraints for each $(i+1)$-th row, $(i+1)$-column, and $(i+1)$-th block. `mkDistinct()` is used to add the constraint that all variables inside should be assigned with different values.

For row constraints, we can simply use `p[i]` to refer to all elements: `p[i][0]` … `p[i][8]` in $(i+1)$-th row:

```
solver.add(ctx.mkDistinct(p[i]));
```

For column constraints, we need to explicitly put all elements: `p[0][i]` … `p[8][i]` in $(i+1)$-th column in `mkDistinct()` to make them distinctive from each other:

```
solver.add(ctx.mkDistinct(p[0][i],p[1][i],p[2][i],p[3][i],p[4][i],p[5][i],p[6][i],p[7][i],p[8][i]
));
```

For block constraints, we can imagine that `b[0]` … `b[8]` are the 9 blocks for the Sudoku:

```
b[0]  b[1]  b[2]
b[3]  b[4]  b[5]
b[6]  b[7]  b[8]
```

For each block `b[i]`, the upper-left element is `p[(i/3)*3][(i%3)*3]`, which leads to the following encoding of the blocks constraints:

```
solver.add(ctx.mkDistinct(p[(i/3)*3][(i%3)*3],p[(i/3)*3][(i%3)*3+1],p[(i/3)*3][(i%3)*3+2],p[(i/3)*3+1][(i%3)*3],p[(i/3)*3+1][(i%3)*3+1],p[(i/3)*3+1][(i%3)*3+2],p[(i/3)*3+2][(i%3)*3],p[(i/3)*3+2][(i%3)*3+1],p[(i/3)*3+2][(i%3)*3+2]));
```

2.4 Adding Known Value Constraints:

First, we need to load the file record of the known values.

```
File inFile=new File("./src/cmpt/input.txt");
Scanner in=new Scanner(inFile);
```

The file has 9 lines. Each line *i* contains 9 integers, which corresponds to 9 elements in the *i*-th row of the Sudoku. The integer "0" means that the corresponding element in Sudoku is unknown, while other integer values indicate that the corresponding element in Sudoku is given.

In implementation, we can write loops to read the *j*-th element in the *i*-th row. If the value is not zero, then we need to impose a known value constraint for `p[i][j]`:

```
for(int i=0;i<9;i++) {
      String line[]=in.nextLine().split(" ");
      for(int j=0;j<9;j++){
           int val=Integer.parseInt(line[j]);
           if(val!=0){
               solver.add(ctx.mkEq(p[i][j],ctx.mkInt(val)));
           }
      }
}
```

Since both step 2.3 and step 2.4 requires loops, in actual implementation we can interleave the operations. In the outer *i* loop, we add the distinction constraints, and in the inner *j* loop, we add known value constraints for `p[i][j]`:

```
for(int i=0;i<9;i++) {
      solver.add(ctx.mkDistinct(p[i]));
      solver.add(ctx.mkDistinct(p[0][i],p[1][i],p[2][i],p[3][i],p[4][i],p[5][i],p[6][i],p[7][i],p[8][i]));
      solver.add(ctx.mkDistinct(p[(i/3)*3][(i%3)*3],p[(i/3)*3][(i%3)*3+1],p[(i/3)*3][(i%3)*3+2],p[(i/3)*3+1][(i%3)*3],p[(i/3)*3+1][(i%3)*3+1],p[(i/3)*3+1][(i%3)*3+2],p[(i/3)*3+2][(i%3)*3],p[(i/3)*3+2][(i%3)*3+1],p[(i/3)*3+2][(i%3)*3+2]));
      String line[]=in.nextLine().split(" ");
      for(int j=0;j<9;j++){
          int val=Integer.parseInt(line[j]);
```

```
            if(val!=0){
                solver.add(ctx.mkEq(p[i][j],ctx.mkInt(val)));
            }
        }
}
```

2.5 Output:

So far, we have added all the constraints for the Sudoku problem, and now we can use Z3 solver to check the satisfiability.

First, create a file to store the results.

```
File outFile=new File("./src/cmpt/output.txt");
FileWriter out=new FileWriter(outFile);
```

Then, check if the constraints are satisfiable:

```
Status status=solver.check();
```

Then, use `status.toString()` to see the result. If the constraints are satisfiable, then use `solver.getModel()` and `model.getConstInterp()` to obtain one feasible assignments of the variables and save the results in the file. If the constraints are not satisfiable, then simply output "No solution" in the file.

```
if(status.toString().equals("SATISFIABLE")){
        Model model = solver.getModel();
        for(int i=0;i<9;i++){
            for(int j=0;j<9;j++){
                out.write(model.getConstInterp(p[i][j]).toString());
                out.write(" ");
            }
            out.write("\n");
        }
}
else if(status.toString().equals("UNSATISFIABLE")){
        out.write("No Solution");
        out.write("\n");
}
in.close();
out.close();
```